Beyond Cryptanalysis is Software Security the Next Threat for Smart Cards

Jean-Louis Lanet

Beyond Cryptanalysis is Software Security the Next Threat for Smart Cards

Jean-Louis Lanet

INRIA, LHS-PEC
263 Avenue Général Leclerc, 35042 Rennes,
jean-louis.lanet@inria.fr
http://secinfo.msi.unilim.fr/lanet/

Abstract. Smart cards have been considered for a long time as a secure container for storing secret data and executing programs that manipulate them without leaking any information. In the last decade, a new form of attack that uses the hardware has been intensively studied. We have proposed in the past to pay attention also to easier attacks that use only software. We demonstrated through several proof of concepts that such an approach should be a threat under some hypotheses. We have been able to execute self-modifying code, return address programming and so on. More recently we have been able to retrieve secret keys belonging to another application. Then all the already published attacks should have been a threat but the industry increased the counter measures to mitigate for each of the published attack. In such a sensitive domain, we always submit the attacks to the industrial partners but also national agencies before publishing any attack. Within such an approach, they have been able to patch their system before any vulnerabilities should be exploited.

Keywords: Smart Card, Attacks, Ethical Process

1 Introduction

Java Card is a kind of smart card that implements one of the two editions, "*Classic Edition*" or "*Connected Edition*", of the standard Java Card 3.0 [12]. Such a smart card embeds a virtual machine which interprets codes already romized with the operating system or downloaded after issuance. Due to security reasons, the ability to download code into the card is controlled by a protocol defined by Global Platform [7]. This protocol ensures that the owner of the code has the necessary authorization to perform the action. Java Card is an open platform for smart cards, *i.e.* able of loading and executing new applications after issuance. Thus, different applications from different providers run in the same smart card. Thanks to type verification, byte codes delivered by the Java compiler and the converter (in charge of giving a compact representation of class files) are safe, *i.e.* the loaded application is not hostile to other applications in the Java Card. Furthermore, the Java Card firewall checks permissions between applications in

the card, enforcing isolation between them.

Java Cards have shown an improved robustness compared to native applications regarding many attacks. They are designed to resist to numerous attacks using both physical and logical techniques. Currently, the most powerful attacks are hardware based attacks and particularly fault attacks. A fault attack modifies parts of memory content or signal on internal bus and lead to deviant behavior exploitable by an attacker. A comprehensive consequence of such attacks can be found in [11]. Although fault attacks have been mainly used in the literature from a cryptanalytic point of view (see [1, 9, 13]), they can be applied to every code layers embedded in a device. For instance, while choosing the exact byte of a program the attacker can bypass counter-measures or logical tests.

The design of a Java Card virtual machine cannot rely on the environmental hypotheses of Java. In fact, physical attacks have never been taken into account during the design of the Java platform. To fill this gap, card designers developed an interpreter which relies on the principle that once the application has been linked to the card, it will not be modifiable again. The trade-off is between a highly defensive virtual machine which will be too slow to operate and an offensive interpreter that will expose too much vulnerabilities. The know-how of a smart card design is in the choice of a set of minimal counter-measures with high fault coverage.

Nevertheless some attacks have been successful in retrieving secret data from the card. Thus we will present here a survey of different approaches to get access to data, which should bypass Java security components. The aim of an attacker is to generate malicious applications which can bypass firewall restrictions and modify other applications, even if they do not belong to the same security package. Several papers were published and they differ essentially on the hypotheses of the platform vulnerabilities. After a brief presentation of the Java Card platform and its security functions, we will present attacks based on a faulty implementation of the transaction, due to ambiguities in the specification. Then we will describe the flaws that can be exploited with an ill-typed applet and we will finish with hostile applet that gain privilege to access the physical processor leading to the dump of the operating system and the crypto API.

2 Smart Card Security

Smart cards security depends on the underlying hardware and the embedded software. Embedded sensors (light sensors, heat sensors, voltage sensors, *etc.*) protect the card from physical attacks. While the card detects such an attack, it has the possibility to erase quickly the content of the EEPROM preserving the confidentiality of secret data or blocking definitely the card (Card is mute). In addition to the hardware protection, softwares are designed to securely ensure that application are syntactically and semantically correct before installation

and also sometimes during execution. They also manage sensitive information and ensure that the current operation is authorized before executing it. The Byte Code Verifier guarantees type correctness of code, which in turn guarantees the Java properties regarding memory access. For example, it is impossible in Java to perform arithmetic on reference. Thus, it must be proved that the two elements on top of the stack are of primitive types before performing any arithmetic operation. On the Java platform, byte code verification is invoked at load time by the loader. Due to the fact that Java Card does not support dynamic class loading, byte code verification is performed at installation time *i.e.* before loading the Card APplet (CAP) onto the card. However, most of the Java Card smart cards do not have an on-card BCV as it is quite expensive in terms of memory consumption. Thus, a trusted third party performs an off-card byte code verification and sign it, and on card its digital signature is checked.

Moreover, the Firewall performs checks at runtime to prevent applets from accessing (reading or writing) data of other applets. When an applet is created, the system uses a unique applet identifier (AID) from which it is possible to retrieve the name of the package in which it is defined. If two applets are instances of classes coming from the same Java Card package, they are considered belonging to the same context. The firewall isolates the contexts in such a way that a method executing in one context cannot access any attribute or method of objects belonging to another context unless it explicitly exposes functionality *via* a Shareable Interface Object.

Smart card security is a complex problem with different points of view but products based on Java Card Virtual Machine (JCVM) have passed successfully real-world security evaluations for major industries around the world. It is also the platform that has passed high level security evaluations for issuance by banking associations and by leading government authorities, they have also achieved compliance with FIPS 140-1 certification scheme. Nevertheless implementations have suffered severals attacks either hardware or software based. Some of them succeeded into getting access to the EEPROM (code of the downloaded applets) but as far as we know nobody succeeded into reversing the code *i.e.* having access to the code of the virtual machine, the operating system and the cryptographic algorithm implementations. These latter are protected by the interpretation layer which denies access to other memories than the EEPROM.

3 Some Software Attacks again Java Card

3.1 Ambiguity in the specification: the type confusion

Erik Poll made a presentation at CARDIS'08 about attacks on smart cards. In his paper [10], he did a quick overview of the classical attacks available on smart cards and gave some counter-measures. He explained the different kinds of attacks and the associated counter-measures. He described four methods (1) CAP file manipulation, (2) Fault injection, (3) Shareable interfaces mechanisms abuse

and (4) Transaction Mechanisms abuse.

He proposed a new way to abuse the Transaction mechanism (4). The purpose of transaction is to make a group of operations becomes atomic. Of course, it is a widely used concept, like in databases, but still hard to implement. By definition, the rollback mechanism should also deallocate any objects allocated during an aborted transaction, and reset references to such objects to null. However, Erik Poll find some strange cases where the card keep the references of objects allocated during transaction even after a roll back.

If he can get the same behavior, it should be easy to get and exploit type confusion. A first example is to get two arrays of different types, for example a byte and a short array. One of them is a field (permanent storage) the second is a local variable. While aborting the transaction, the permanent reference must be nullified. But the specification do not explain what to do with local variables if they reference also a permanent object. Poll discovered that some cards cleared all the references while other let dangling pointers. In such a case reallocating the memory will let the dangling pointer referencing another object of potentially another type. If he declares a byte array of 10 bytes, and he has another reference as a short array, he will be able to read 10 shorts, so 20 bytes. With this method he can read the 10 bytes saved after the array. If he increases the size of the array, he can read as much memory as he wants. The main problem is more how to read memory before the array. The other confusion he used is an array of bytes and an object. If he puts a byte as first object attribute, it is bound to the array length. It is then really easy to change the length of the array using the reference to the object.

3.2 Weakness in the linker process

The Java Card Specification defines the linking step done during the loading of CAP file. When the software is downloading in the card, the Java Card Virtual Machine provides a way to link, the CAP file to install, with the installed Java Card API. This step is done thanks to a tokens link resolution references in the `Constant Pool` component. To friendly find where each token is used, the `Reference Location` component keeps a list of offsets, in the `Method` Component. So, in this loading step, the JCVM translates, with the help of the `Constant Pool` component and the `Reference Location` component, each reference to methods or fields use in the CAP file. To abuse the linking mechanism [14], [8] we modify the token following any natural instructions, as `invokestatic`, which are following by a token. If the card have not any BCV component, a modification may push the linked reference on the stack and returned at the end of the current function.

Using this approach we are able to use the on board linker to generate the correct information, to store it on top of the stack and to send it back to the reader. Thanks to this information leakage we are able to obtain all the linked

address of the Java Card API for a given card. For retrieving one address we need to build one CAP file. Retrieving the complete API, need to generate 98 test cases for the methods of the classes and 60 test cases for the interfaces. All the test cases are valid whatever the card is tested. It means that the effort to design the test cases for retrieving the addresses will be reusable on all the cards. This attack is completely generic and independent of the platform.

3.3 Dumping the EEPROM

As said previously, the verifier must check several points. In particular: there are no violations of memory management and any stack underflow or overflow. This means that these checks are potentially not verified during run time and then can lead to vulnerabilities. The Java frame is a non persistent data structure but can be implemented in different manners and the specification gives no design direction for it. Getting access to the RAM provides information of other objects like the APDU buffer, return address of a method and so on. So, changing the return of a local address modifies the control flow of the call graph and returns it to a specific address.

The EMAN2 attack [3] allows to modify the value of the return address of a method by storing a short into a local. By choosing the right value for the local number we overwrite the return address. In a given card the return address register is stored at MAX_LOCAL + 2. The value stored in this register will be the address where Java PC will be updated while returning from the current method. We just need to define a static array which is stored close to the method area. Then after returning from the method, the JCVM will execute the content of the array. Due to the fact that `getstatic` and `putstatic` are not checked by the firewall, we can read the content of the memory. The shell code is presented in Listing 1.1.

Listing 1.1. Executing the basic shell code

```
7C 01 00        getStatic   0x0100
78              sreturn
```

This code puts on top of the stack, the content of the memory at the address 0x0100 and returns this value. The caller has just to store it into the APDU buffer and the value is send to the terminal. Then, the third byte of the static array must be incremented and the next call will return the value of the address 0x0101. We just need to manage the carry from the low byte to the high byte representing the address. Another way to update the return address is the `sinc` instruction. The `sinc` instruction aims to increase a local short variable by a constant value given in its parameter.

Recently, Faugeron [6] presented a way to fool the Java Card runtime based on the `dup_x` instruction. This instruction duplicates the top of operands stack

words and inserts them below. This instruction takes two parameters encoded on 1-byte where the high nibble describes the number of words to duplicate and the low nibble defines where the duplicated words are placed. Since the Java Card operands stack does not contain enough elements, the runtime uses the system data as words for the `dup_x` instruction. Thus, an attacker can shift the value of the frame header by a custom words pushed on the stack.

3.4 Dumping the ROM

In [4] we demonstrated the ability to dump the content of the ROM and thus to get access to the implementation of the cryptographic functions. We used several weaknesses. During the analysis of EEPROM dump corresponding to a linked applets into the smart card memory, a method with an abnormal call has been noticed at the address `0xDBE6`. This address corresponds to another EEPROM address and not a ROM address. At that address we found a table which corresponded to non standard method headers. The JCVM Specification [5] defines a method as a `method_header_info`, described in the listing 1.2, and its associated byte code.

Listing 1.2. Java Card Method Header Info

```
method_header_info {
  u1 bitfield {
    bit[4] flags // a mask of modifiers defined for the method
    bit[4] max_stack // max cells required during execution of
                     // the method
  }
  u1 bitfield {
    bit [4] nargs // number of parameters passed to the method
    bit [4] max_locals // number of local variables declared
                       // by the method
  }
}
```

For the flag value, three defined possibilities are expected:

- 0x0: it is a normal method;
- 0x8 (`ACC_EXTENDED`): the method represents an extended method;
- 0x4 (`ACC_ABSTRACT`): the method represents an abstract method;
- All other flag values are reserved.

Each methods of the table contains a non standardized flag value (*i.e.* : 2). Moreover, the associated byte code (1-byte) cannot be an instruction. On the other side, we also also have a set of interesting values in the EEPROM. We assumed that all these values are addresses that refer to the ROM, except one which refers to the EEPROM. To prove our hypothesis we checked the data contained at the address corresponds to a 8051 assembler language which corresponds to the native code for the targeted card. We reversed the code in order

to verify the calling convention of this native Java Interface.

To exploit this weakness, we added to the method table a `fake method` (a method with a flag value equals to `2`) contains an offset to an address in the indirection table. Each element in the indirection table refers to a native function. At this offset we put the address of our shellcode. Without integrity check, the Java Card Runtime execute the malicious code. Finally, to execute the native shell code the parameter of an `invokestatic` instruction, or another kind of call instruction should be changed by the address of our `fake method`. Thus, the faulty instruction provides a way to execute any shell code with native privilege. With this shell code, we have been able to do a memory dump of the ROM code. Examining carefully the code we discovered the cryptographic code corresponding to the embedded algorithms within this specific card.

3.5 A complete Methodology to Attack Smart Card

In his PhD, Bouffard [2] applied the Attack Tree Analysis (ATA) to have a global view on the vulnerability of the smart card. Attack trees have been introduced by Schneier in[15], they represent a convenient approach to analyze the different ways in which a system can be attacked. It is an analytical technique (top-down) where an undesirable event is defined and the system is then analyzed to find the combinations of basic events that could lead to the undesirable event. Such an analysis is closed to the risk analysis community with the cause-effect diagrams. An attack tree is a tree in which the nodes represent attacks. The root node of the tree is the property that an attacker wants to break. Children of a node are refinements of this goal, and leafs therefore represent initial causes. An attack tree is not a model of all possible combination but a restricted set. It is related to the property evaluated. In this case, code integrity is the most sensible property because if not guaranteed, it enables the attacker to execute any arbitrary code.

The property we want to protect is the integrity of the code which can be violated by a Control Flow Transfer (CFT) attack. So one of the events which can transgress this property is the CFT attack which becomes the root of the subtree of the code integrity ATA. Until now, the control flow attack instance was only the EMAN2 attack. To mitigate such an attack, it was only required to either check at runtime the locals, pass the BCV or enable a frame integrity check. Such leaf requires to check the underflow of the stack on some instructions. Some of the cards now implement a frame integrity that disallows to arbitrary write into the frame. One can remark that the Frame Integrity detection mechanism covers both EMAN2 and Faugeron's attack, while the Check of Local Variables covers only the EMAN2.

To succeed, detection event and mitigation event must be inhibited with a not gate. In this figure a `nand` gate plays this role. The CFT attack represented in Figure 1 will succeed if the adequate ill formed CAP is loaded and no integrity check or no local variable check are present on the card and the BCV is bypassed.

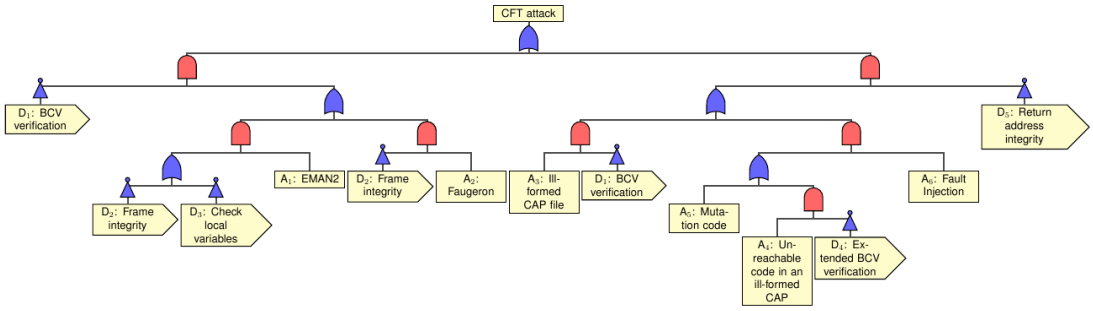

Fig. 1. Attack Tree

When the event is detected, then the card is muted and the attack is stopped. We use this methodology to provide a clear overview on how different events can be combined to set up attacks that can break the integrity of the code. We do not pay attention here on the valuation of the effort of the attacker but on the efficiency of a counter measure. The minimal cut of an ATA defines the minimal sets of basic events determining an attack scenario. Closer to the root is the detection event or the mitigation event better is the coverage.

4 Conclusion and Future Works

We have presented here a set of attacks concerning the smart card world an in particular the Java Card domain. The abality to download application from an untrusted environment open the possibility to characterize the content of the smart card. In particular it allows the attacker to recover code from application (EEPROM) or from the system (ROM) but also to recover some of the data that do not belong him. integrity and confidentiality can be broken just using the techniques used in main stream IT programming. We proposed a methodology based on attack trees to model the knowledge of the attacker. By defining a minimal cut in such a tree, we define the scenario that could lead to the attack. Such a tree can also be used as a defensive means by defining close to the root the adequate counter measure. This optimize the coverage and thus the efficiency of the defense.

References

1. Aumller, C., Bier, P., Fischer, W., Hofreiter, P., Seifert, J.P.: Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures. In: Kaliski, B., Ko, e., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2002, Lecture Notes in Computer Science, vol. 2523, pp. 81–95. Springer Berlin / Heidelberg (2003)

2. Bouffard, G.: A Generic Approach for Protecting Java Card Smart Card Against Software Attacks. Ph.D. thesis, University of Limoges, 123 Avenue Albert Thomas, 87060 LIMOGES CEDEX (Oct 2014)
3. Bouffard, G., Iguchi-Cartigny, J., Lanet, J.L.: Combined Software and Hardware Attacks on the Java Card Control Flow. In: CARDIS, pp. 283–296 (2011)
4. Bouffard, G., Lanet, J.L.: Reversing the operating system of a java based smart card. Journal of Computer Virology and Hacking Techniques 10(4), 239–253 (2014), http://dx.doi.org/10.1007/s11416-014-0218-7
5. Card, J.: 2.1. 1 virtual machine specification. SUN Microsystems Inc (2000)
6. Faugeron, E.: Manipulating the Frame Information With an Underflow Attack. In: Francillon, A., Rohatgi, P. (eds.) CARDIS. Lecture Notes in Computer Science, vol. 8419. Springer (2013)
7. GlobalPlatform: Card Specification. GlobalPlatform Inc., 2.2.1 edn. (Jan 2011)
8. Hamadouche, S., Bouffard, G., Lanet, J.L., Dorsemaine, B., Nouhant, B., Magloire, A., Reygnaud, A.: Subverting byte code linker service to characterize java card api. In: Seventh Conference on Network and Information Systems Security (SAR-SSI). pp. 75–81 (May 22rd to 25th 2012)
9. Hemme, L.: A differential fault attack against early rounds of (triple-) DES. Cryptographic Hardware and Embedded Systems-CHES 2004 pp. 170–217 (2004)
10. Hubbers, E., Poll, E.: Transactions and non-atomic API calls in Java Card: specification ambiguity and strange implementation behaviours. Tech. rep., University of Nijmegen (2004)
11. Iguchi-Cartigny, J., Lanet, J.L.: Developing a Trojan applets in a Smart Card. Journal in Computer Virology 6, 343–351 (2010)
12. Oracle: Java Card 3 Platform, Virtual Machine Specification, Classic Edition. No. Version 3.0.4, Oracle, Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065 (2011)
13. Piret, G., Quisquater, J.: A differential fault attack technique against SPN structures, with application to the AES and KHAZAD. Cryptographic Hardware and Embedded Systems-CHES 2003 pp. 77–88 (2003)
14. Razafindralambo, T., Bouffard, G., Lanet, J.: A friendly framework for hidding fault enabled virus for Java based smartcard. In: Nora Cuppens-Boulahia, Frdric Cuppens, J.G.A. (ed.) Data and Applications Security and Privacy XXVI, Lecture Notes in Computer Science, vol. 7371, pp. 122–128. Springer Berlin / Heidelberg (2012)
15. Schneier, B.: Attack trees: Modeling security threat. In: Dr. Dobbs journal (1999)